1

Programming Basics and Strings

This chapter is a gentle introduction to the practice of programming in Python. Python is a very rich language with many features, so it is important to learn to walk before you learn to run. Chapters 1 through 3 provide a basic introduction to common programming ideas, explained in easily digestible paragraphs with simple examples.

If you are already an experienced programmer interested in Python, you may want to read this chapter quickly and take note of the examples, but until Chapter 3 you will be reading material with which you've probably already gained some familiarity in another language.

If you are a novice programmer, by the end of this chapter you will have learned some guiding principles for programming, as well as directions for your first interactions with a programming language — Python. The exercises at the end of the chapter provide hands-on experience with the basic information that you'll have learned.

How Programming Is Different from Using a Computer

The first thing you need to understand about computers when you're programming is that you control the computer. Sometimes the computer doesn't do what you expect, but even when it doesn't do what you want the first time, it should do the same thing the second and third time — until you take charge and change the program.

The trend in personal computers has been away from reliability and toward software being built on top of other, unreliable, software. The results that you live with might have you believing that computers are malicious and arbitrary beasts, existing to taunt you with unbearable amounts of extra work and various harassments while you're already trying to accomplish something. If you do feel this way, you already know that you're not alone. However, after you've learned how to program, you gain an understanding of how this situation has come to pass, and perhaps you'll find that you can do better than some of the programmers whose software you've used.

Note that programming in a language like Python, an **interpreted** language, means that you are not going to need to know a whole lot about computer hardware, memory, or long sequences of 0s and 1s. You are going to write in text form like you are used to reading and writing but in a different and simpler language. Python is the language, and like English or any other language(s) you speak, it makes sense to other people who already speak the language. Learning a programming language can be even easier, however, because programming languages aren't intended for discussions, debates, phone calls, plays, movies, or any kind of casual interaction. They're intended for giving instructions and ensuring that those instructions are followed. Computers have been fashioned into incredibly flexible tools that have found a use in almost every business and task that people have found themselves doing, but they are still built from fundamentally understandable and controllable pieces.

Programming Is Consistency

In spite of the complexity involved in covering all of the disciplines into which computers have crept, the basic computer is still relatively simple in principle. The internal mechanisms that define how a computer works haven't changed a lot since the 1950s when transistors were first used in computers.

In all that time, this core simplicity has meant that computers can, and should, be held to a high standard of consistency. What this means to you, as the programmer, is that anytime you tell a computer to metaphorically jump, you must tell it how high and where to land, and it will perform that jump — over and over again for as long as you specify. The program should not arbitrarily stop working or change how it works without you facilitating the change.

Programming Is Control

Programming a computer is very different from creating a program, as the word applies to people in real life. In real life, we ask people to do things, and sometimes we have to struggle mightily to ensure that our wishes are carried out — for example, if we plan a party for 30 people and assign two of them to bring the chips and dip and two of them to bring the drinks.

With computers that problem doesn't exist. The computer does exactly what you tell it to do. As you can imagine, this means that you must pay some attention to detail to ensure that the computer does just what you want it to do.

One of the goals of Python is to program in **blocks** that enable you to think about larger and larger projects by building each project as pieces that behave in well-understood ways. This is a key goal of a programming style known as **object-oriented programming.** The guiding principle of this style is that you can create reliable pieces that still work when you piece them together, that are understandable, and that are useful. This gives you, the programmer, control over how the parts of your programs run, while enabling you to extend your program as the problems you're solving evolve.

Programming Copes with Change

Programs are run on computers that handle real-world problems; and in the real world, plans and circumstances frequently change. Because of these shifting circumstances, programmers rarely get the opportunity to create perfectly crafted, useful, and flexible programs. Usually, you can achieve only two of these goals. The changes that you will have to deal with should give you some perspective and lead you to program cautiously. With sufficient caution, you can create programs that know when they're being asked to exceed their capabilities, and they can fail gracefully by notifying their users that they've stopped. In the best cases, you can create programs that explain what failed and why. Python offers especially useful features that enable you to describe what conditions may have occurred that prevented your program from working.

What All That Means Together

Taken together, these beginning principles mean that you're going to be introduced to programming as a way of telling a computer what tasks you want it to do, in an environment where you are in control. You will be aware that sometimes accidents can happen and that these mistakes can be accommodated through mechanisms that offer you some discretion regarding how these conditions will be handled, including recovering from problems and continuing to work.

The First Steps

First, you should go online to the web site for the book, following the procedure in the Introduction, and follow the instructions there for downloading PythonCard. PythonCard is a set of utilities that provides an environment for programming in Python. PythonCard is a product that's free to use and distribute and is tailor-made for writing in Python. It contains an editor, called codeEditor, that you will be using for the first part of this book. It has a lot in common with the editor that comes with Python, called *idle,* but in the opinion of the authors, codeEditor works better as a teaching tool because it was written with a focus on users who may be working on simpler projects. In addition, codeEditor is a program written in Python.

Programs are written in a form called *source code*. Source code contains the instructions that the language follows, and when the source code is read and processed, the instructions that you've put in there become the actions that the computer takes.

Just as authors and editors have specialized tools for writing for magazines, books, or online publications, programmers also need specialized tools. As a starting Python programmer, the right tool for the job is codeEditor.

Starting codeEditor

Depending on your operating system, you will start codeEditor in different ways.

Once it is installed on your system with PythonCard, on Linux or Unix-based systems, you can just type **codeEditor** in a terminal or shell window and it will start.

On Windows, codeEditor should be in your Start menu under Programs ⇨ PythonCard. Simply launching the program will get you started.

When you start codeEditor for the first time, it doesn't display an open file to work with, so it gives you the simplest possible starting point, a window with very little in it. Along the left side, you'll see line numbers. Programmers are often given information by their programs about where there was a problem, or where something happened, based on the line number in the file. This is one of the features of a good programming editor, and it makes it much easier to work with programs.

Using codeEditor's Python Shell

Before starting to write programs, you're going to learn how to experiment with the Python shell. For now, you can think of the Python shell as a way to peer within running Python code. It places you inside of a running instance of Python, into which you can feed programming code; at the same time, Python will do what you have asked it to do and will show you a little bit about how it responds to its environment. Because running programs often have a **context** — things that you as the programmer have tailored to your needs — it is an advantage to have the shell because it lets you experiment with the context you have created. Sometimes the context that you're operating in is called your **environment.**

Try It Out Starting the Python Shell

To start the Python shell from codeEditor, pull down the Shell menu in the codeEditor's menu bar and select Shell window. This will open a window with the Python shell in it (no surprises here) that just has simple text, with line numbers along the left side (see Figure 1-1). You can get a similar interface without using PythonCard by starting the regular Python interpreter, without PythonCard's additions, by just typing **python** on a Unix system or by invoking Python from the Start menu on a Windows system.

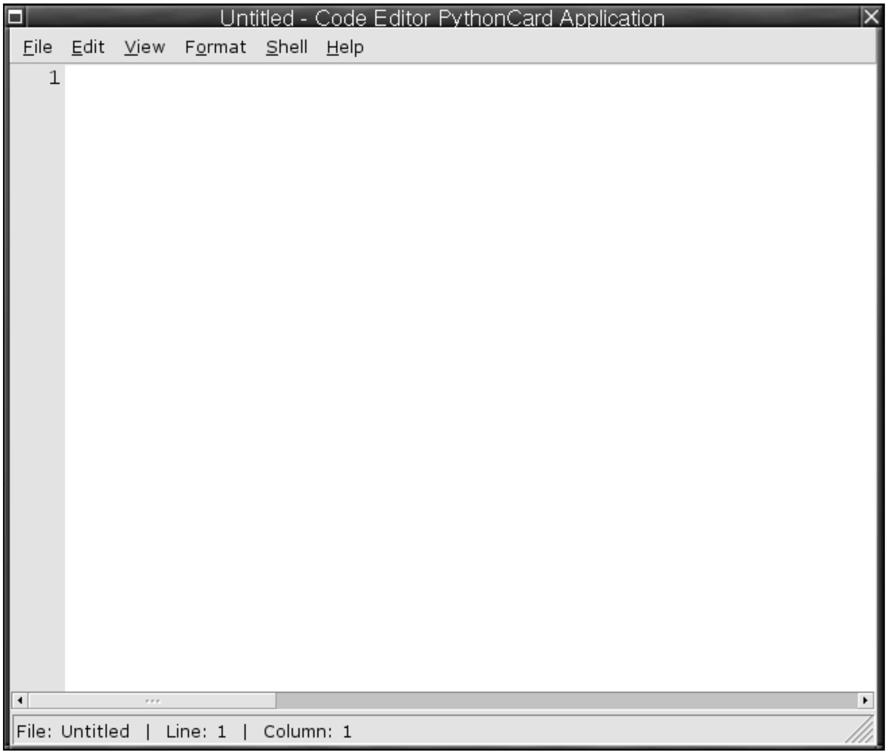
Untitled - Code Editor PythonCard Application
File Edit View Format Shell Help
1
File: Untitled | Line: 1 | Column: 1

Figure 1-1

After you've started the shell, you'll be presented with some information that you don't have to be concerned about now (from, import, pcapp, and so on), followed by the sign that the interpreter is ready and waiting for you to work with it: `>>>`.

```
>>> import wx
>>> from PythonCard import dialog, util
>>> bg = pcapp.getCurrentBackground()
>>> self = bg
>>> comp = bg.components
>>>
```

How It Works

The codeEditor is a program written in Python, and the Python shell within it is actually a special programming environment that is enhanced with features that you will use later in the book to help you explore Python. The `import`, `from`, and other statements are covered in Chapter 7 in depth, but for now they're not important.

Beginning to Use Python — Strings

At this point, you should feel free to experiment with using the shell's basic behavior. Type some text, in quotes; for starters, you could type the following:

```
>>> "This text really won't do anything"
"This text really won't do anything"
>>>
```

You should notice one thing immediately: After you entered a quote ("), codeEditor's Python shell changed the color of everything up to the quote that completed the sentence. Of course, the preceding text is absolutely true. It did nothing: It didn't change your Python environment; it was merely *evaluated* by the running Python instance, in case it did determine that in fact you'd told it to do something. In this case, you've asked it only to read the text you wrote, but doing this doesn't constitute a change to the environment.

However, you can see that Python indicated that it saw what you entered. It showed you the text you entered, and it displayed it in the manner it will always display a string — in quotes. As you learn about other **data types,** you'll find that Python has a way of displaying each one differently.

What Is a String?

The **string** is the first data type that you're being introduced to within Python. Computers in general, and programming languages specifically, segregate everything they deal with into types. Types are categories for things within a program with which the program will work. After a thing has a type, the program (and the programmer) knows what to do with that thing. This is a fundamental aspect of how computers work, because without a named type for the abstract ideas that they work with, the computer won't know how to do basic things like combine two different values. However, if you have two things, and they're of the same type, you can define easy rules for combining them. Therefore, when the type of a thing has been confirmed, Python knows what its options are, and you as the programmer know more about what to do with it.

Why the Quotes?

Now, back to strings in particular. Strings are the basic unit of text in Python. Unlike some other programming languages, a single letter is represented as a one-letter string. Instead of trying to explain strings in terms of other concepts in a vacuum, let's create some examples of strings using the Python shell and build from there.

Try It Out Entering Strings with Different Quotes

Enter the following strings, keeping in mind the type of quotes (single or double) and the ends of lines (use the Enter key when you see that the end of a line has been reached):

```
>>> "This is another string"
'This is another string'
>>> 'This is also a string'
'This is also a string'
>>> """This is a third string that is some
...     how different"""
'This is a third string that is some\n    how different'
```

How It Works

If you use different quotes, they may look different to you; to the Python interpreter; however all of them can be used in the same situations and are very similar. For more information, read on.

These examples raise a few questions. In your first text example, you saw that the text was enclosed in double quotes, and when python saw two quotes it repeated those double quotes on the next line. However, in the preceding example, double quotes are used for "This is another string", but below it single quotes are used. Then, in the third example, three double quotes in a row are used, and after the word "some" we used the Enter key, which caused a new line to appear. The following section explains these seemingly arbitrary conventions.

Understanding Different Quotes

Three different types of quotes are used in Python. First, there are the single and double quotes, which you can look at in two ways. In one way, they are identical. They work the same way and they do the same things. Why have both? Well, there are a couple of reasons. First, strings play a huge part in almost any program that you're going to write, and quotes define strings. One challenge when you first use them is that quotes aren't special characters that appear only in computer programs. They are a part of any normal English text to indicate that someone has spoken. In addition, they are used for emphasis or to indicate that something is literally what was seen or experienced.

The dilemma for a programming language is that when you're programming, you can only use characters that are already on a keyboard. However, the keys on a keyboard can be entered by the average user, so obviously people normally use those keys for tasks other than programming! Therefore, how do you make it a special character? How do you indicate to the language that you, the programmer, mean something different when you type a set of quotes to pass a string to your program, versus when you, as the programmer, enter quotes to explain something to the person using your program?

One solution to this dilemma is a technique that's called **escaping.** In most programming languages, at least one character, called an **escape character,** is designated; and it has the power to remove the special significance from other special characters, such as quotes. This character in Python is the backslash (\). Therefore, if you have to quote some text within a string and it uses the same style of quote in which you enclosed the entire string, you need to escape the quote that encloses the string to prevent Python from thinking that it has prematurely reached the end of a string. If that sounds confusing, it looks like this:

```
>>> 'And he said \'this string has escaped quotes\''
"And he said 'this string has escaped quotes'"
```

Returning to those three examples, normally a running Python shell will show you a string that it has evaluated in single quotes. However, if you use a single quote within a string that begins and ends with double quotes, Python will display that string with double quotes around it to make it obvious to you where the string starts and where it ends:

```
>>> 'Ben said "How\'re we supposed to know that?"'
'Ben said "How\'re we supposed to know that?"'
>>>
```

This shows you that there is no difference between single and double quoted strings. The only thing to be aware of is that when you start a string with a double quote, it can't be ended by a single quote, and vice versa. Therefore, if you have a string that contains single quotes, you can make your life easier by enclosing the string in double quotes, and vice versa if you've got strings with quotes that have been enclosed in single quotes. SQL, the language that is used to obtain data from databases, will often have single quoted strings inside of them that have nothing to do with Python. You can learn more about this when you reach Chapter 14. One more important rule to know is that by themselves, quotes will not let you create a **newline** in a string. The newline is the character that Python uses internally to mark the end of a line. It's how computers know that it's time to start a new line.

> **Within strings, Python has a way of representing special characters that you normally don't see — in fact, that may indicate an action, such as a newline, by using sequences of characters starting with a backslash (\). (Remember that it's already special because it's the escape character and now it's even more special.) The newline is `\n`, and it is likely the most common special character you will encounter.**
>
> **Until you see how to print your strings, you'll still see the escaped characters looking as you entered them, as `\n`, instead of, say, an actual line ending, with any more tests starting on the next line.**

Python has one more special way of constructing strings, one that will almost always avoid the entire issue of requiring an escape character and will let you put in new lines as well: the triple quote. If you ever use a string enclosed in three quotes in a row — either single or double quotes, but all three have to be the same kind — then you do not have to worry about escaping any single instance of a single or double quote. Until Python sees three of the same quotes in a row, it won't consider the string ended, and it can save you the need to use escape characters in some situations:

```
>>> """This is kind of a special string, because it violates some
...     rules that we haven't talked about yet"""
"This is kind of a special string, because it violates some\n    rules that we
haven't talked about yet"
```

As you can see here, Python enables you to do what you want in triple-quoted strings. However, it does raise one more question: What's that `\n` doing there? In the text, you created a new line by pressing the Enter key, so why didn't it just print the rest of the sentence on another line? Well, Python will provide an interpretation to you in the interest of accuracy. The reason why `\n` may be more accurate than showing you the next character on a new line is twofold: First, that's one way for you to tell Python that you're interested in printing a new line, so it's not a one-way street. Second, when displaying this kind of data, it can be confusing to actually be presented with a new line. Without the `\n`, you may not know whether something is on a new line because you've got a newline character or because there are spaces that lead up to the end of the line, and the display you're using has wrapped around past the end of the current line and is continued on the next line. By printing `\n`, Python shows you exactly what is happening.

Putting Two Strings Together

Something that you are probably going to encounter more than a few times in your programming adventures is multiple strings that you want to print at once. A simple example is when you have separate records of a person's first name and last name, or their address, and you want to put them together. In Python, each one of these items can be treated separately, as shown here:

```
>>> "John"
'John'
>>> "Q."
'Q.'
>>> "Public"
'Public'
>>>
```

Try It Out Using + to Combine Strings

To put each of these distinct strings together, you have a couple of options. One, you can use Python's own idea of how strings act when they're added together:

```
>>> "John" + "Q." + "Public"
'JohnQ.Public'
```

How It Works

This does put your strings together, but notice how this doesn't insert spaces the way you would expect to read a person's name; it's not readable, because using the plus sign doesn't take into account any concepts of how you want your string to be presented.

You can easily insert spaces between them, however. Like newlines, spaces are characters that are treated just like any other character, such as A, s, d, or 5. Spaces are not removed from strings, even though they can't be seen:

```
>>> "John" + " " + "Q." + " " + "Public"
'John Q. Public'
```

After you determine how flexible you need to be, you have a lot of control and can make decisions about the format of your strings.

Putting Strings Together in Different Ways

Another way to specify strings is to use a **format specifier.** It works by putting in a special sequence of characters that Python will interpret as a placeholder for a value that will be provided by you. This may initially seem like it's too complex to be useful, but format specifiers also enable you to control what the displayed information looks like, as well as a number of other useful tricks.

Try It Out Using a Format Specifier to Populate a String

In the simplest case, you can do the same thing with your friend, John Q.:

```
>>> "John Q. %s" % ("Public")
'John Q. Public'
```

How It Works

That `%s` is the format specifier for a string. Several other specifiers will be described as their respective types are introduced. Each specifier acts as a placeholder for that type in the string; and after the string, the `%` sign outside of the string indicates that after it, all of the values to be inserted into the format specifier will be presented there to be used in the string.

You may be wondering why the parentheses are there. The parentheses indicate to the string that it should expect to see a **sequence** that contains the values to be used by the string to populate its format specifiers.

Sequences are a very important part of programming in Python, and they are covered in some detail later. For now, we are just going to use them. What is important to know at this point is that every format specification in a string has to have an element that matches it in the sequence that's provided to it. The items we are putting in the sequence are strings that are separated by commas (if there is more than one). If there is only one, as in the preceding example, the sequence isn't needed, but it can be used.

The reason why this special escape sequence is called a **format specifier** is because you can do some other special things with it — that is, rather than just insert values, you can provide some specifications about how the values will be presented, how they'll look.

Try It Out More String Formatting

You can do a couple of useful things when formatting a simple string:

```
>>> "%s %s %10s" % ("John", "Q.", "Public")
'John Q.     Public'
>>> "%-10s %s %10s" % ("John", "Q.", "Public")
'John       Q.     Public'
```

How It Works

In the first string, the reason why `Public` is so alone along the right side is because the third format specifier in the main string, on the left side, has been told to make room for something that has 10 characters. That's what the `%10s` means. However, because the word `Public` only has 6 characters, Python padded the string with space for the remaining four characters that it had reserved.

In the second string, the `Q.` is stranded in the middle, with `John` and `Public` far to either side. The behavior on its right-hand side has just been explained. The behavior on its left happens for very similar reasons. An area with 10 spaces has been created in the string, but this string was specified with a `%-10s`. The `-` in that specifier means that the item should be pushed to the left, instead of to the right, as it would normally.

Displaying Strings with Print

Up until now, you have seen how Python represents the strings you type, but only how it represents them internally. However, you haven't actually done anything that your program would show to a user. The point of the vast majority of programs is to present users with information — programs produce everything from sports statistics to train schedules to web pages to automated telephone voice response units. The key point is that they all have to make sense to a person eventually.

Try It Out Printing Text with Print

For displaying text, a special feature is built into useful languages, one that helps the programmer display information to users. The basic way to do this in Python is by using the `print` function:

```
>>> print "%s %s %10s" % ("John", "Q.", "Public")
John Q.     Public
>>>
```

You'll notice that there are no longer any quotes surrounding the first, middle, and last name. In this case, it's significant — this is the first thing that you've done that would actually be seen by someone using a program that you've written!

How It Works

`print` is a **function** — a special name that you can put in your programs that will perform one or more tasks behind the scenes. Normally, you don't have to worry about how it happens. (When you start writing your own functions in Chapter 5, you'll naturally start to think more about how this works.)

In this case, the `print` function is an example of a **built-in function,** which is a function included as a part of Python, as opposed to a function that you or another programmer has written. The `print` function performs **output** — that is, it presents something to the user using a mechanism that they can see, such as a terminal, a window, a printer, or perhaps another device (such as a scrolling LED display). Related routines perform **input,** such as getting information from the user, from a file, from the network, and so on. Python considers these input/output (I/O) routines. I/O commonly refers to anything in a program that prints, saves, goes to or from a disk, or connects to a network. You will learn more about I/O in Chapter 8.

Summary

In this chapter, you've begun to learn how to use the programming editor codeEditor, which is a program written in Python for the purpose of editing Python programs. In addition to editing files, codeEditor can run a Python shell, where you can experiment with simple Python programming language statements.

Within the shell, you have learned the basics of how to handle strings, including adding strings together to create longer strings as well as using format specifiers to insert one or more strings into another string that has format specifiers. The format specifier `%s` is used for strings, and it can be combined with numbers, such as `%8s`, to specify that you want space for eight characters — no more and no less. In later chapters, you will learn about other format specifiers that work with other types.

You also learned how to print strings that you have created. Printing is a type of input/output operation (input/output is covered in more detail in Chapter 8). Using the `print` function, you can present users of your program with strings that you have created.

In the next chapter, you will learn about dealing with simple numbers and the operations that you can perform on them, as well as how to combine numbers and strings so that `print` can render numbers displayable. This technique of using format specifiers will enable you to display other types of data as well.

Exercises

1. In the Python shell, type the string, "Rock a by baby,\n\ton the tree top,\t\twhen the wind blows\n\t\t\t the cradle will drop." Experiment with the number and placement of the `\t` and `\n` escape sequences and see how this affects what you see. What do you think will happen?
2. In the Python shell, use the same string indicated in the prior exercise, but display the string using the `print` function. Again, play with the number and location of the `\n` and `\t` escape sequences. What do you think will happen?